AXCIS: Accelerating Architectural Exploration using Canonical Instruction Segments

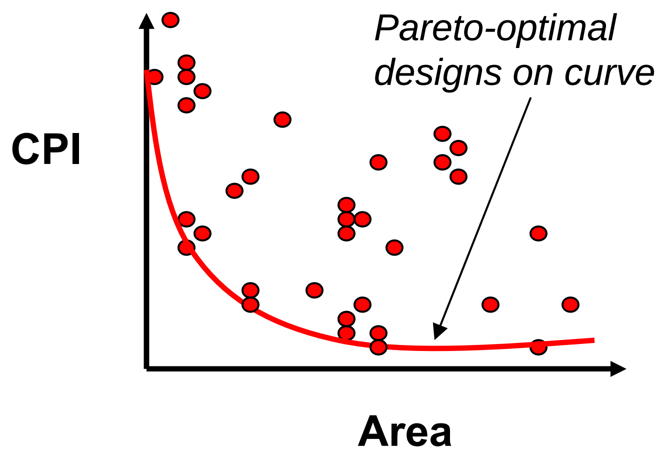
Rose Liu & Krste Asanović

Computer Architecture Group

MIT CSAIL

Simulation for Large Design Space Exploration

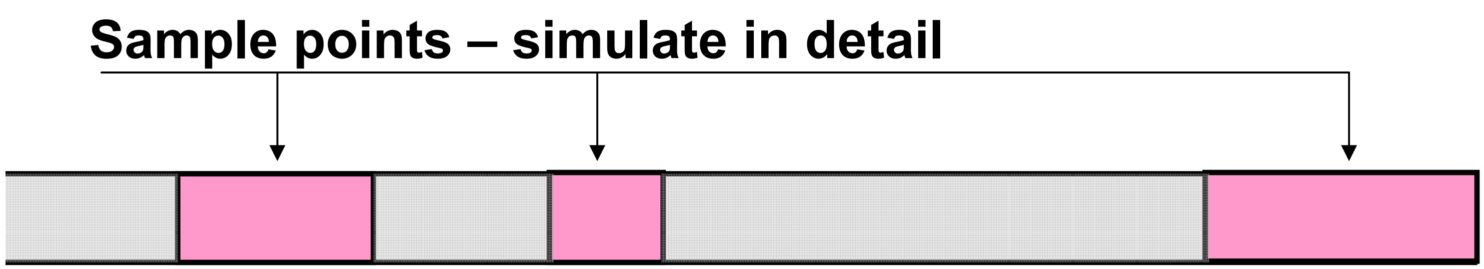
- **Large design space studies explore thousands of processor designs**
  - Identify those that minimize costs and maximize performance



- **Speed vs. Accuracy tradeoff**
  - Maximize simulation speedup while maintaining sufficient accuracy to identify interesting design points for later detailed simulation

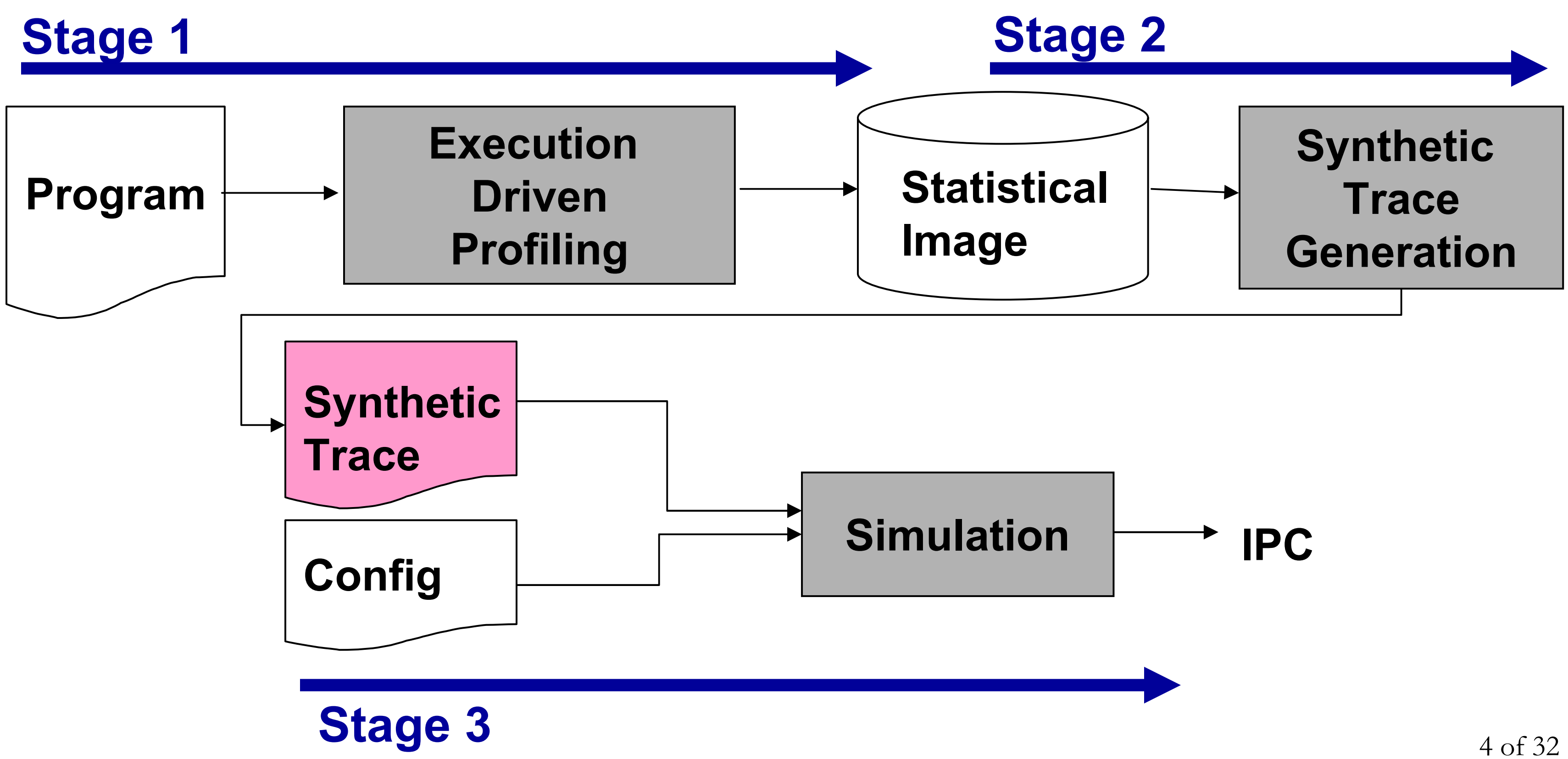
Reduce Simulated Instructions: Sampling

- **Perform detailed microarchitectural simulation during sample points & functional warming between sample points**
  - *SimPoints [ASPLOS, 2002], SMARTS [ISCA, 2003]*
- **Use efficient checkpoint techniques to reduce simulation time to minutes**
  - *TurboSMARTS [SIGMETRICS, 2005], Biesbrouck [HiPEAC, 2005]*

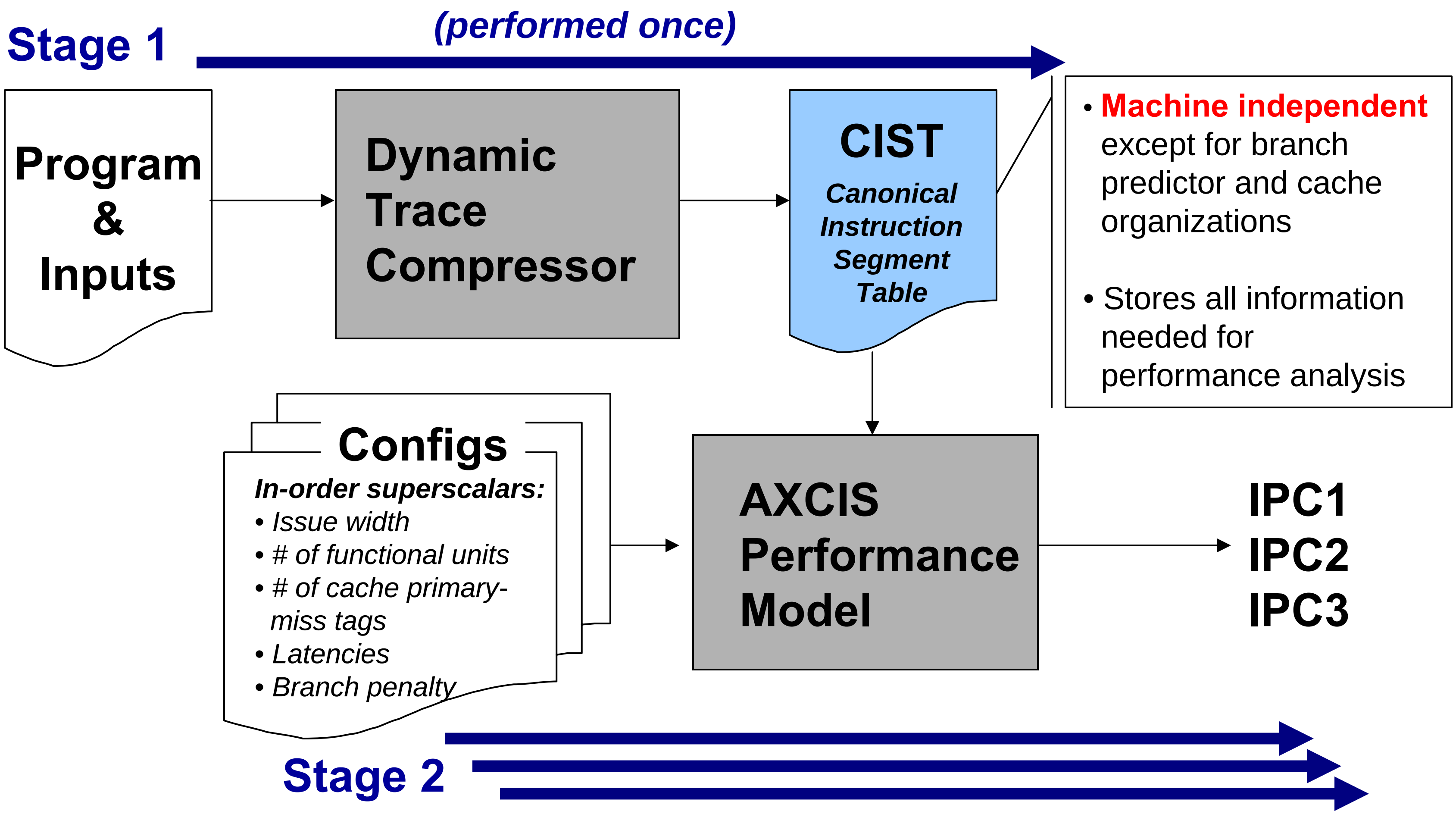


Reduce Simulated Instructions: Statistical Simulation

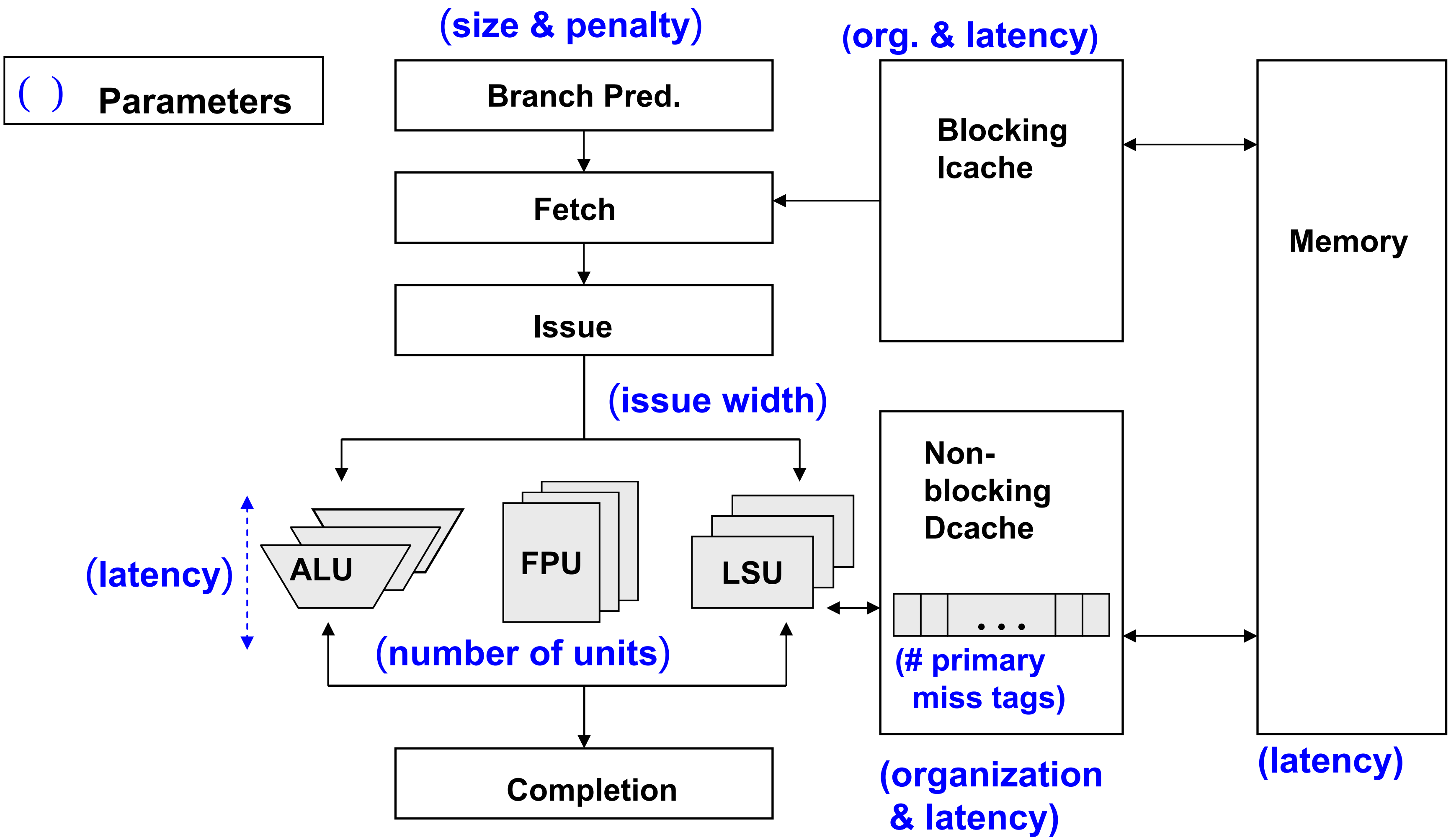
- **Generate a short synthetic trace (with statistical properties similar to original workload) for simulation**
  - *Eeckhout [ISCA, 2004], Oskin [ISCA, 2000] Nussbaum [PACT, 2001]*



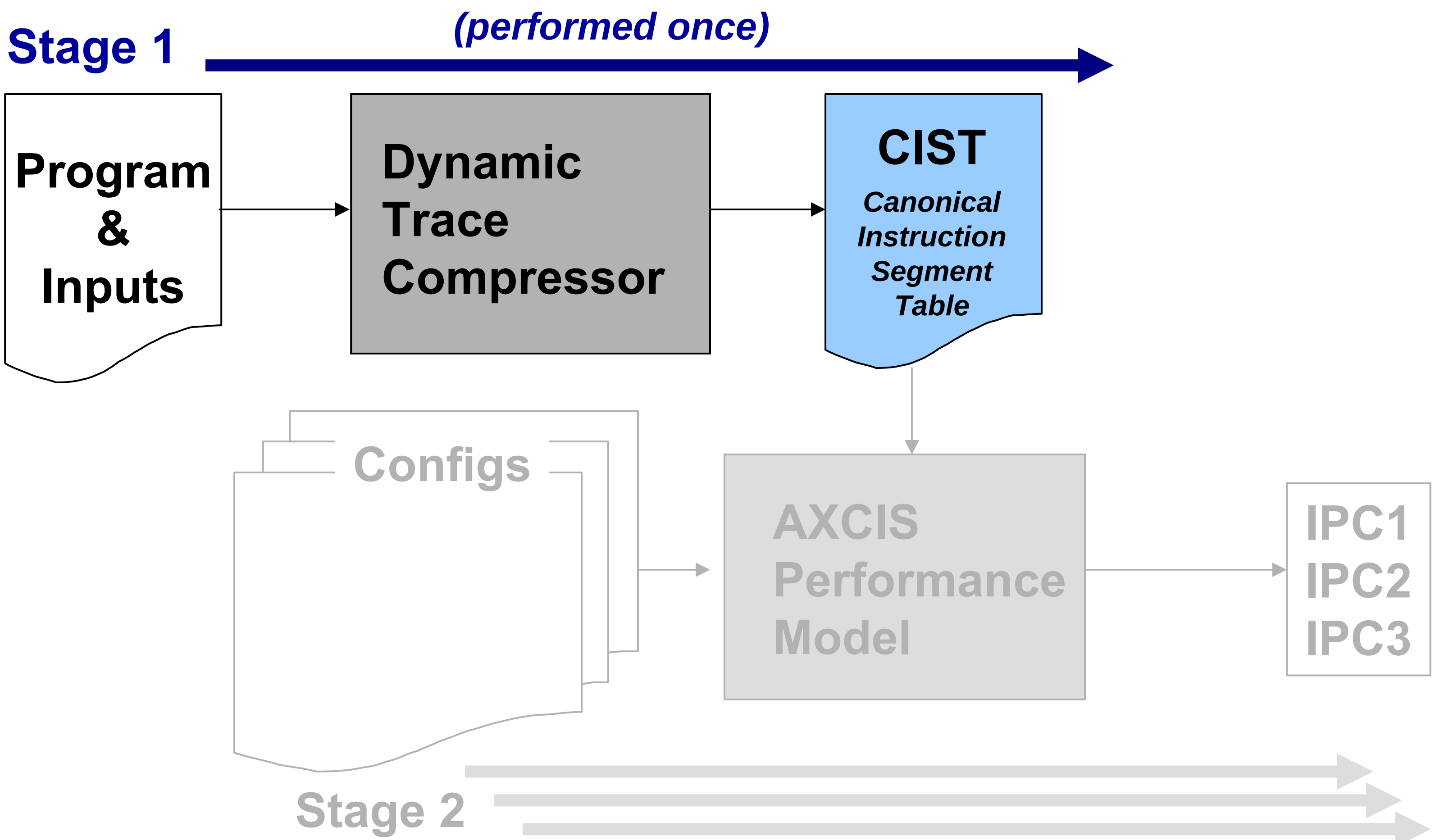
AXCIS Framework



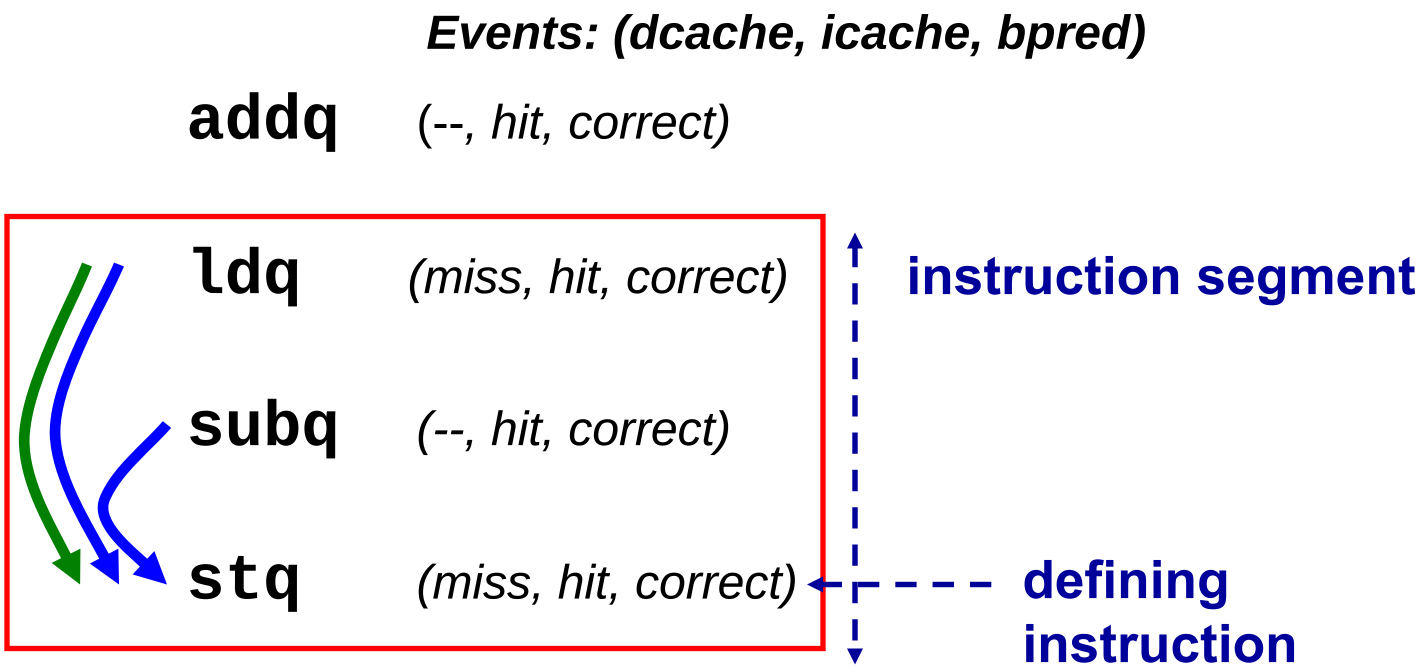
In-Order Superscalar Machine Model



Stage 1: Dynamic Trace Compression

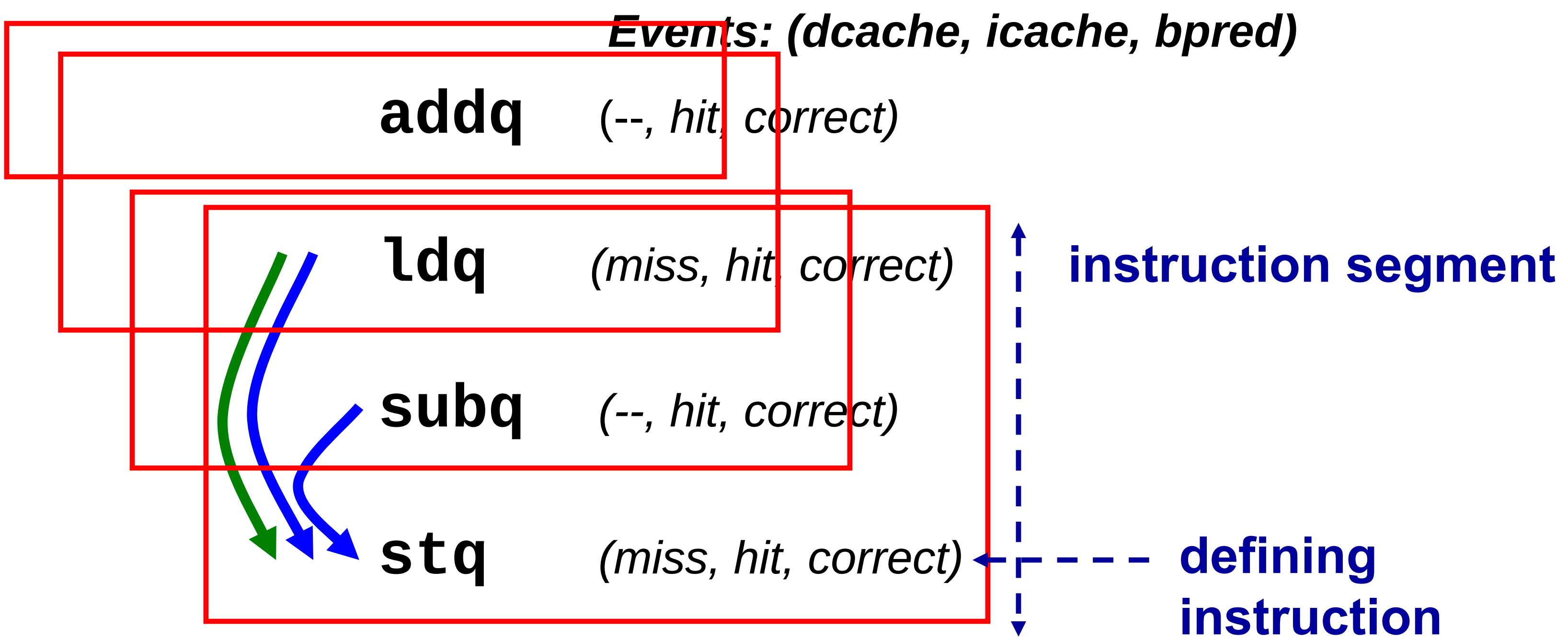


Instruction Segments



- **An instruction segment captures all performance-critical information associated with a dynamic instruction**

Instruction Segments



- An **instruction segment** captures all performance-critical information associated with a dynamic instruction

Dynamic Trace Compression

- **Program behavior repeats due to loops, and repeated function calls**
- **Multiple different dynamic instruction segments can have the same behavior (canonically equivalent) regardless of the machine configuration**

- **Compress the dynamic trace by storing in a table:**
  - 1 copy of each type of segment
  - How often we see it in the dynamic trace

Canonical Instruction Segment Table

addq *(--, hit, correct)*

ldq *(miss, hit, correct)*

addq *(--, hit, correct)*

ldq *(miss, hit, correct)*

subq *(--, hit, correct)*

stq *(miss, hit, correct)*

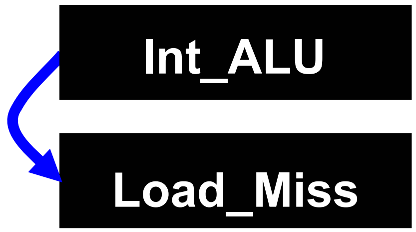
Int_ALU

CIST

Freq	Segment
1	Int_ALU

Canonical Instruction Segment Table

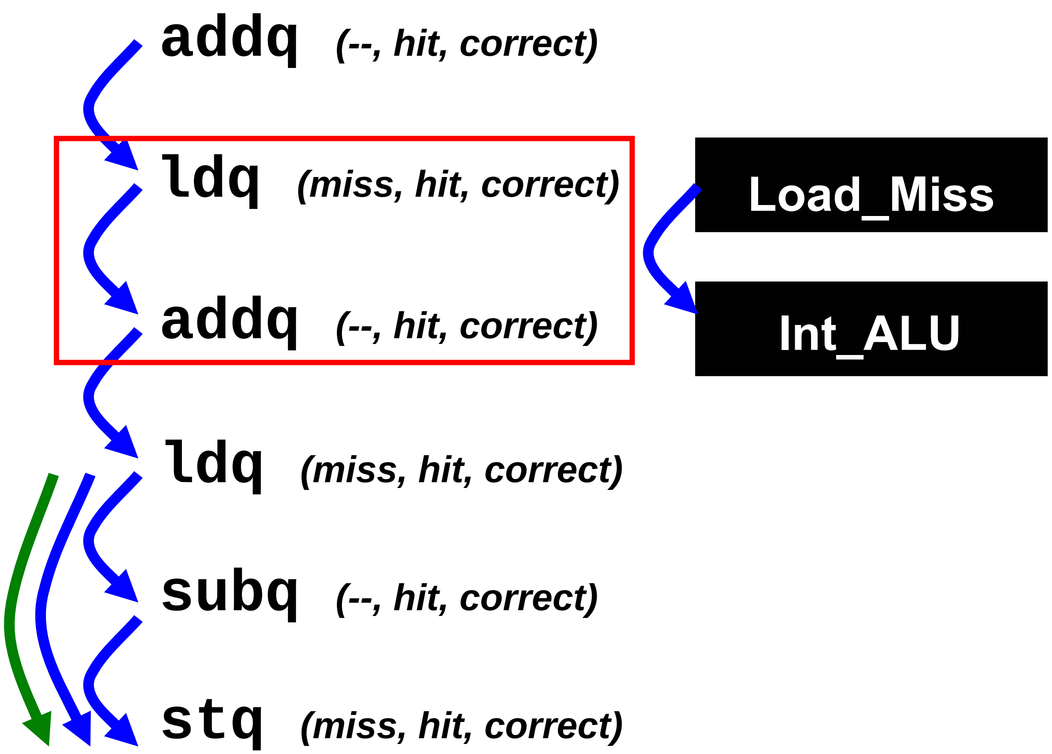
addq *(--, hit, correct)*
ldq *(miss, hit, correct)*
addq *(--, hit, correct)*
ldq *(miss, hit, correct)*
subq *(--, hit, correct)*
stq *(miss, hit, correct)*



CIST

Freq	Segment
1	Int_ALU
1	Int_ALU → Load_Miss

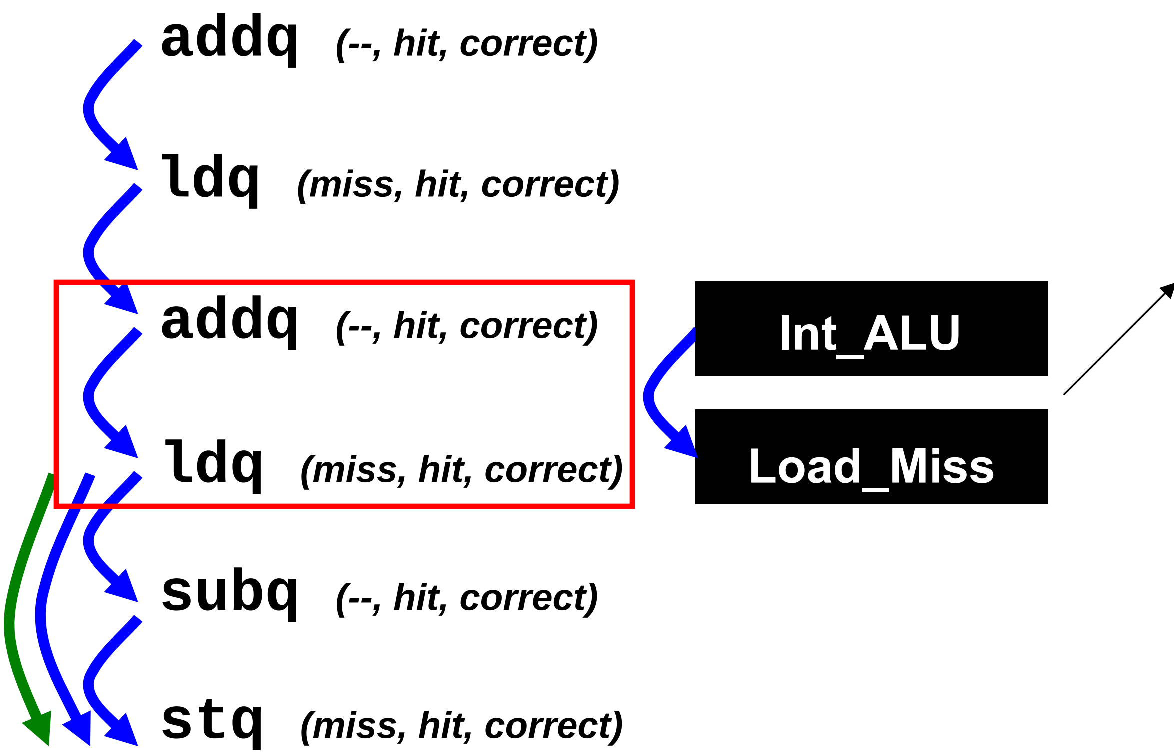
Canonical Instruction Segment Table



CIST

Freq	Segment
1	Int_ALU
1	Int_ALU, Load_Miss
1	Load_Miss, Int_ALU

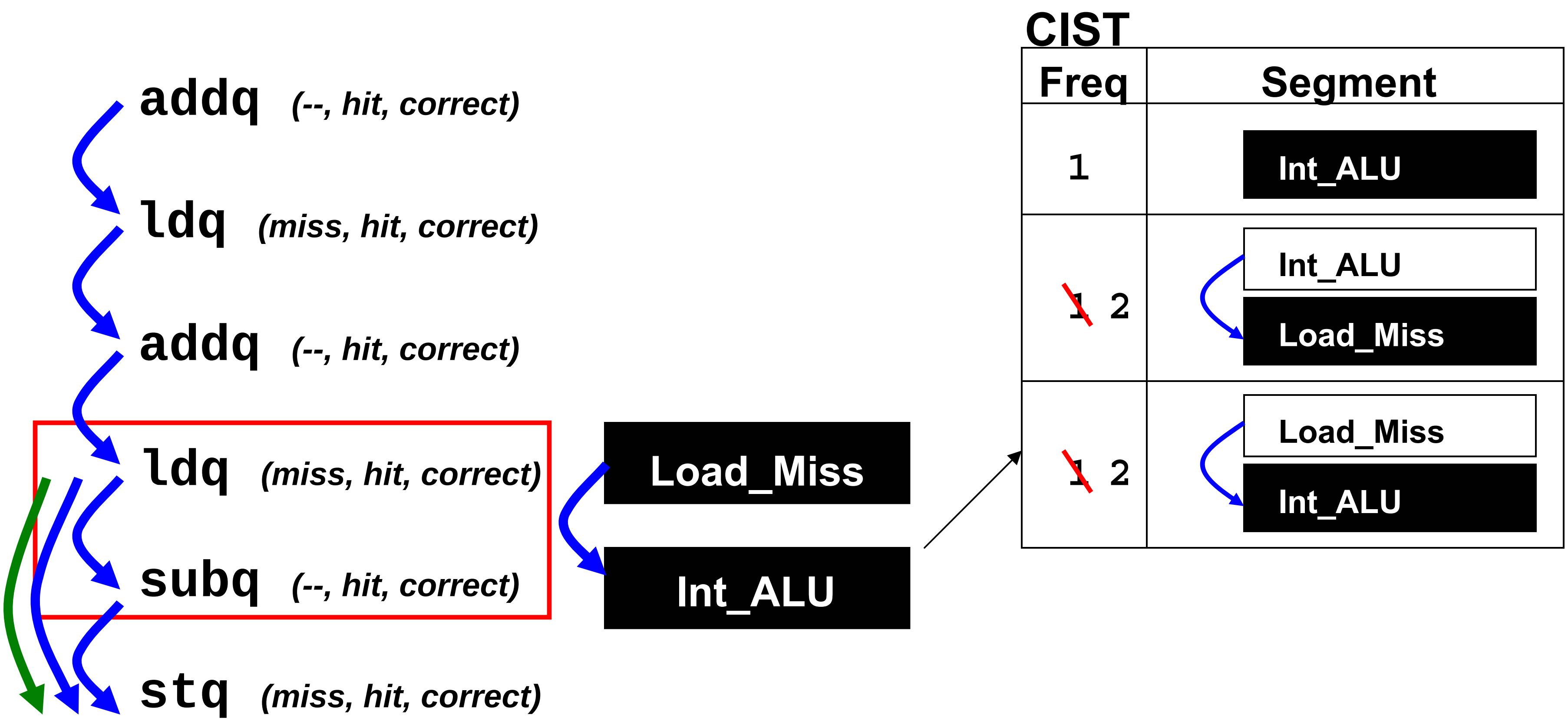
Canonical Instruction Segment Table



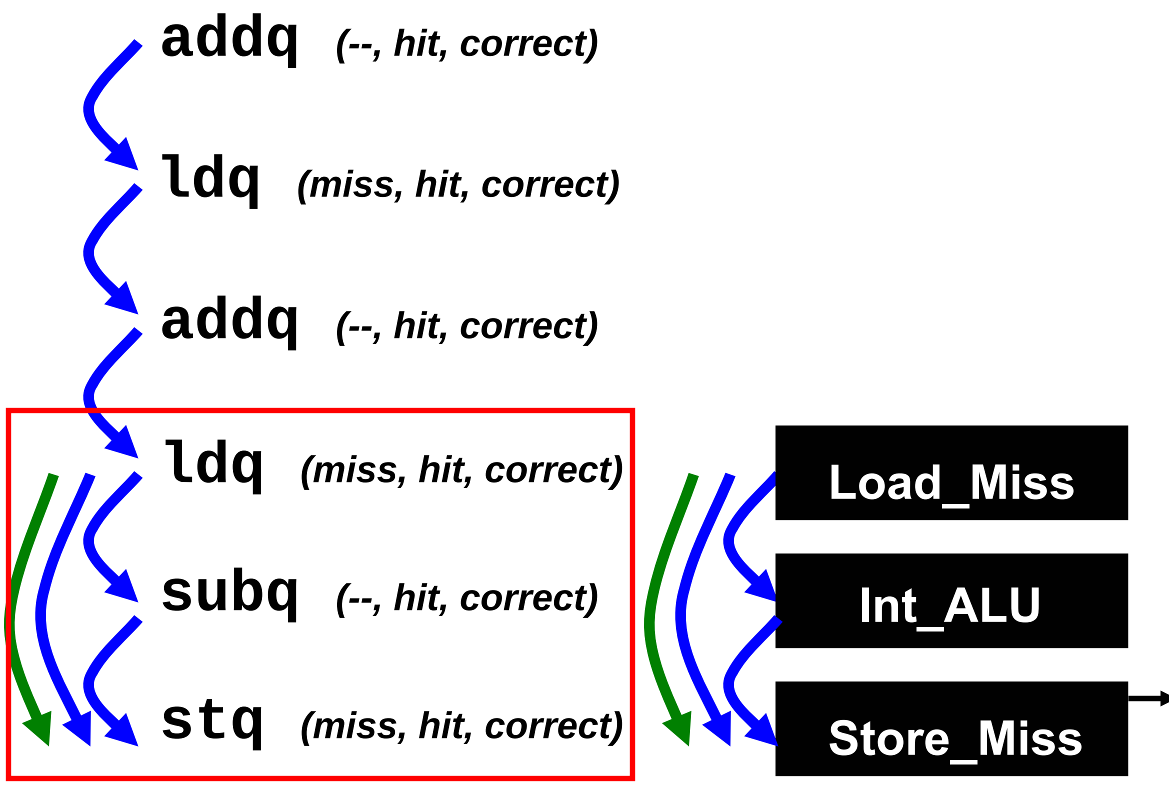
CIST

Freq	Segment
1	Int_ALU
~~1~~ 2	Int_ALU → Load_Miss
1	Load_Miss → Int_ALU

Canonical Instruction Segment Table



Canonical Instruction Segment Table

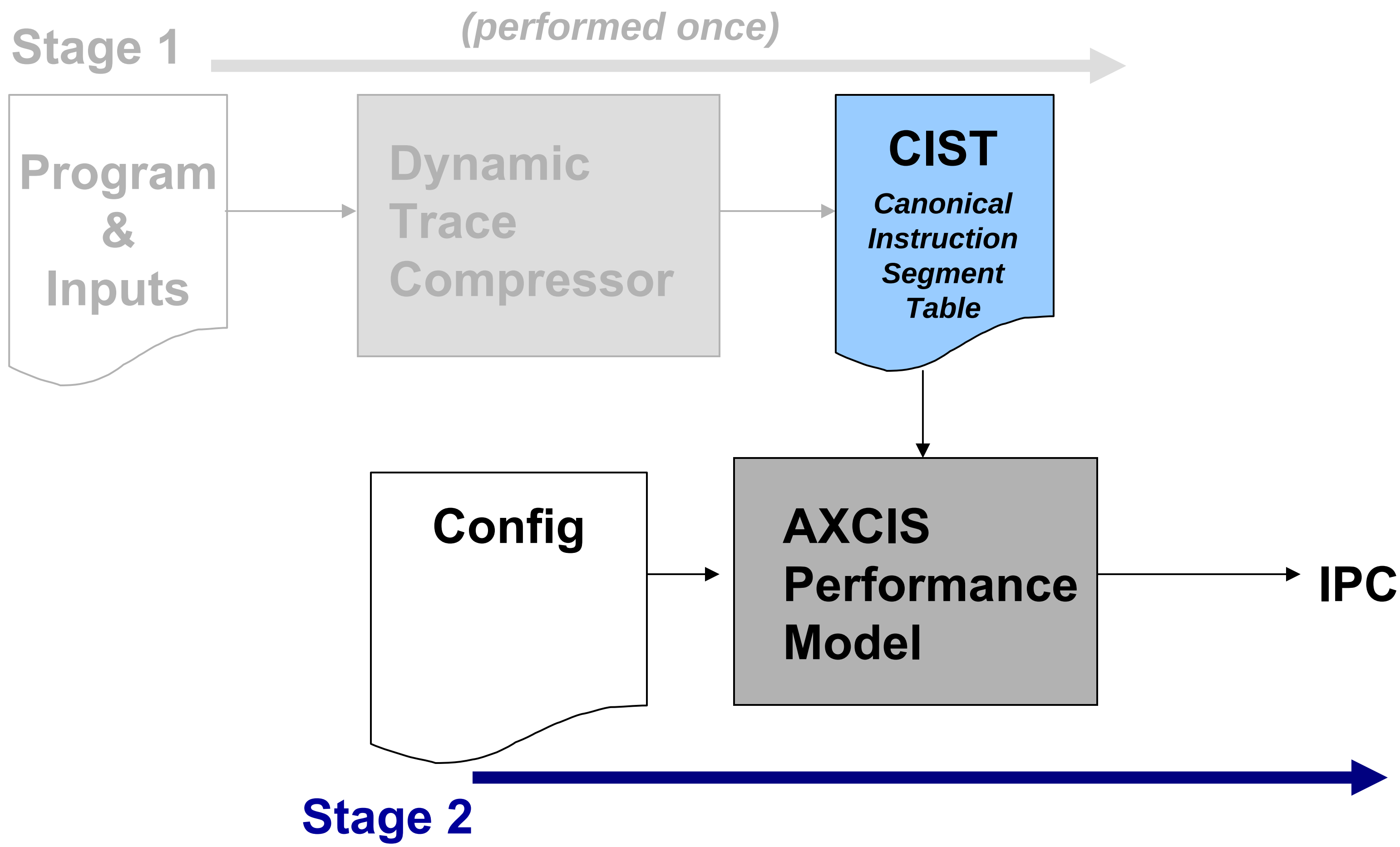


CIST

Freq	Segment
1	Int_ALU
~~1~~ 2	Int_ALU → Load_Miss
~~1~~ 2	Load_Miss → Int_ALU
1	Load_Miss → Int_ALU → Store_Miss

Total ins: 6

Stage 2: AXCIS Performance Model



AXCIS Performance Model

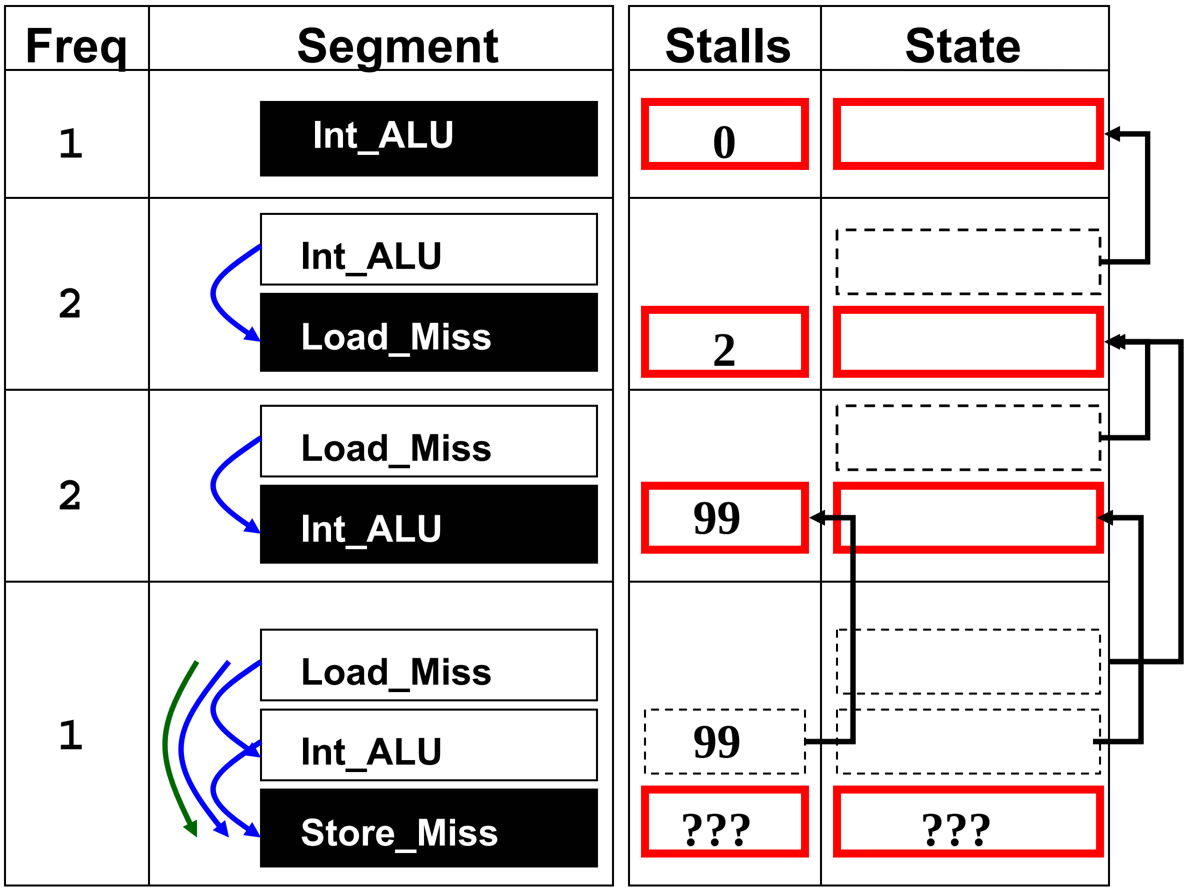
- **Calculates IPC using a single linear dynamic programming pass over the CIST entries**
  - Total work is proportional to the # of CIST entries

$$\textbf{IPC} = \frac{\text{Total Ins}}{\text{Total Cycles}} = \frac{\text{Total Ins}}{\text{Total Ins} + \text{Total Effective Stalls}}$$

$$\textbf{Total Effective Stalls} = \sum_{i=1}^{\text{CIST Size}} \text{Freq(i)} * \text{EffectiveStalls(DefiningIns(i))}$$

$$\textbf{EffectiveStalls} = MAX\,(\,\textbf{stalls}(\text{DataHazards}),\ \textbf{stalls}(\text{StructuralHazards}),\ \textbf{stalls}(\text{ControlFlowHazards})\,)$$

Performance Model Calculations



For each defining instruction:

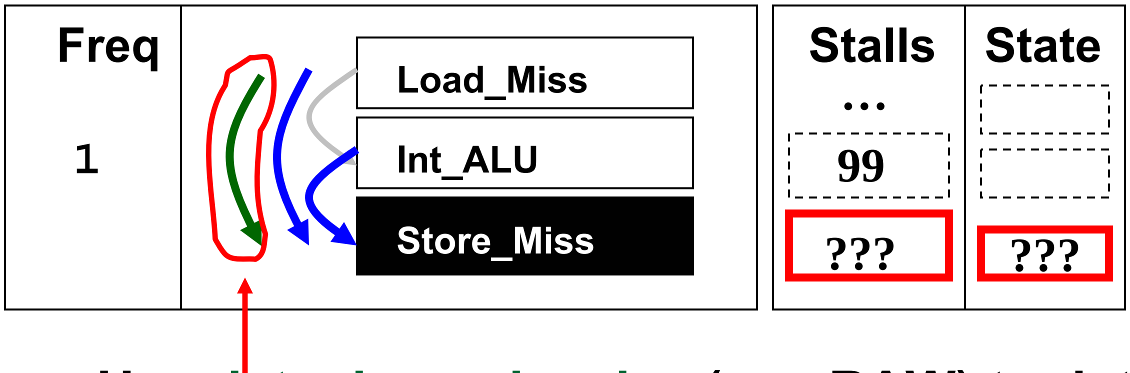
- Calculate its effective stalls & its corresponding microarchitecture state snapshot
- Follow dependencies to look up the effective stalls & state of other instructions in previous entries

Total ins: 6

Look up in previous segment

Calculate

Stall Cycles From Data Hazards



Input configuration:

Ins Type	Latency (cycles)
Int_ALU	3
Load_Miss	100

- **Use data dependencies (e.g. RAW) to detect data hazards**
- **Stalls(DataHazards)**

= *MAX* (-1,
Latency(producer = Load_Miss)
– **DepDist**
– **EffectiveStalls**(IntermediateIns = Int_ALU))

= *MAX* (-1,
(100 – 2 – 99))

= -1 stalls *(can issue with previous instruction)*

Stall Cycles from Structural Hazards

- **CISTs record special dependencies to capture all possible structural hazards across all configurations**
- **The AXCIS performance model follows these special dependencies to find the necessary microarchitectural states to:**
  1. Determine if a structural hazard exists & the number of stall cycles until it is resolved
  2. Derive the microarchitectural state after issuing the current defining instruction

Stall Cycles From Control Flow Hazards

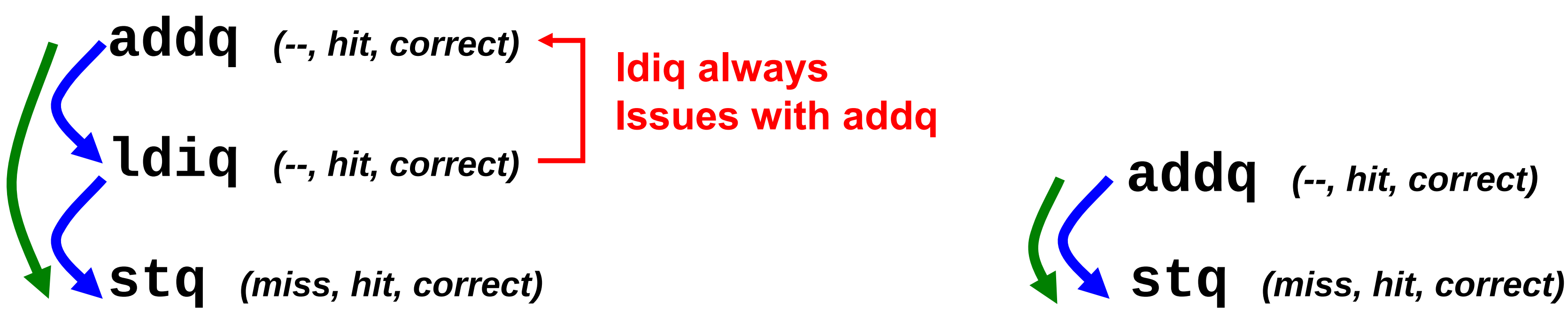
Freq		Icache	Branch Pred.
1	Load_Miss	…	…
	Int_ALU	…	…
	Store_Miss	hit	correct & not taken

- Control flow events directly map to stall cycles

Icache	Bpred	Stalls
Hit	Incorrect & taken/not taken	Mispred penalty
	Correct & taken	0
	Correct & not taken	-1
Miss	Incorrect & taken/not taken	Memory latency + mispred penalty
	Correct & taken	Memory latency
	Correct & not taken	Memory latency - 1

Lossless Compression Scheme

- **Lossless Compression Scheme:** *(perfect accuracy)*
  - Compress two segments if they always experience the same stall cycles regardless of the machine configuration
  - Impractical to implement within the Dynamic Trace Compressor



Three Compression Schemes

- **Instruction Characteristics Based Compression:**
  - Compress segments that "look" alike (i.e. have the same length, instruction types, dependence distances, branch and cache behaviors)
- **Limit Configurations Based Compression:**
  - Compress segments whose defining instructions have the same instruction types, stalls and microarchitectural state under the 2 configurations simulated during trace compression
- **Relaxed Limit Configurations Based Compression:**
  - Relaxed version of the limit-based scheme – does not compare microarchitectural state
  - Improves compression at the cost of accuracy

Experimental Setup

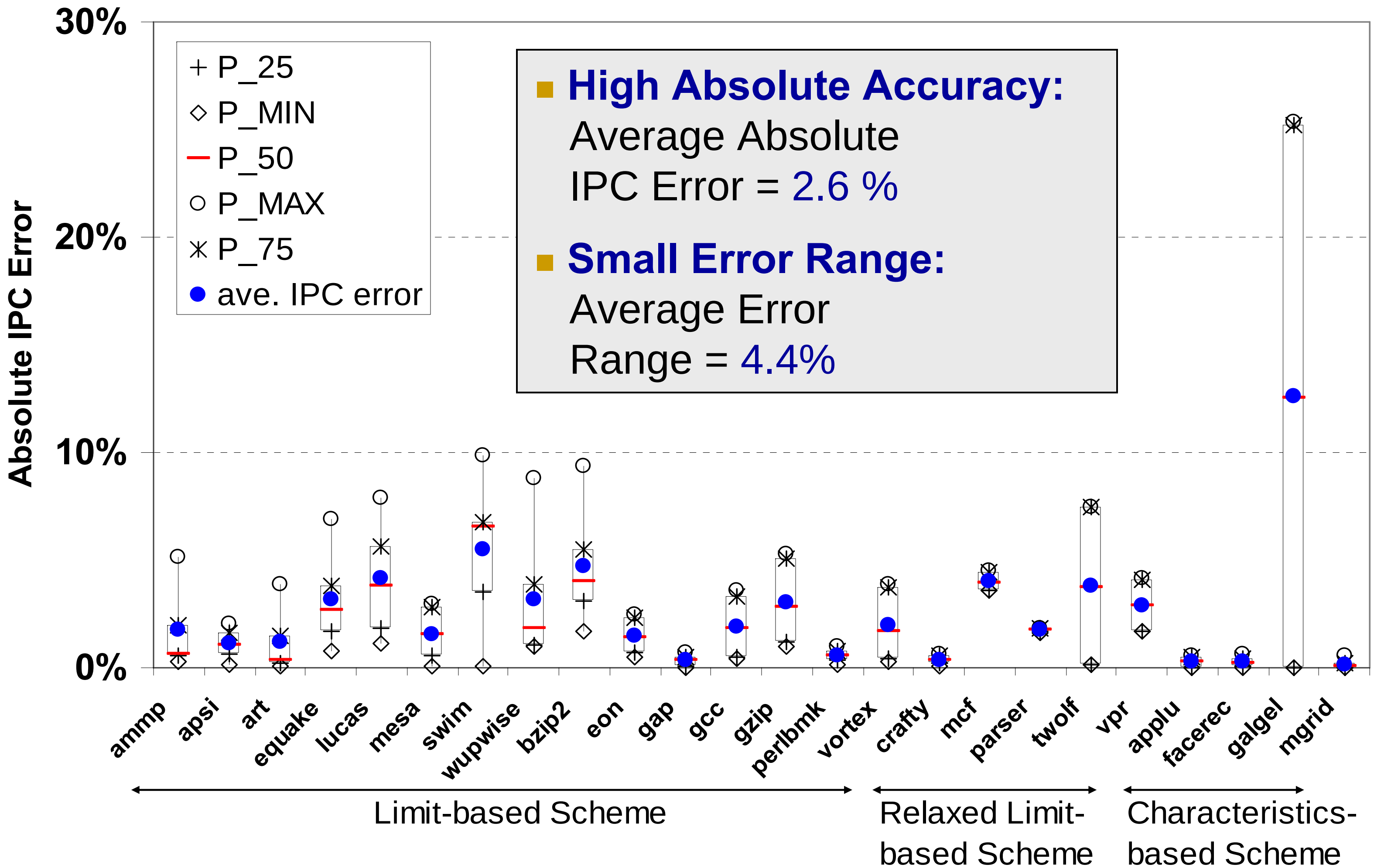
- **Evaluated AXCIS against a baseline cycle accurate simulator on 24 SPEC2K benchmarks**
- **Evaluated AXCIS for:**
  - Accuracy:

$$\text{Absolute IPC Error} = \frac{|\,\text{AXCIS} - \text{Baseline}\,|}{\text{Baseline}} * 100$$

  - Speed: # of CIST entries, time in seconds
- **For each benchmark, simulated a wide range of designs:**
  - Issue width: {1, 4, 8}, # of functional units: {1, 2, 4, 8},
    Memory latency: {10, 200 cycles},
    # of primary miss tags in non-blocking data cache: {1, 8}
- **For each benchmark, selected the compression scheme that provides the best compression given a set accuracy range**

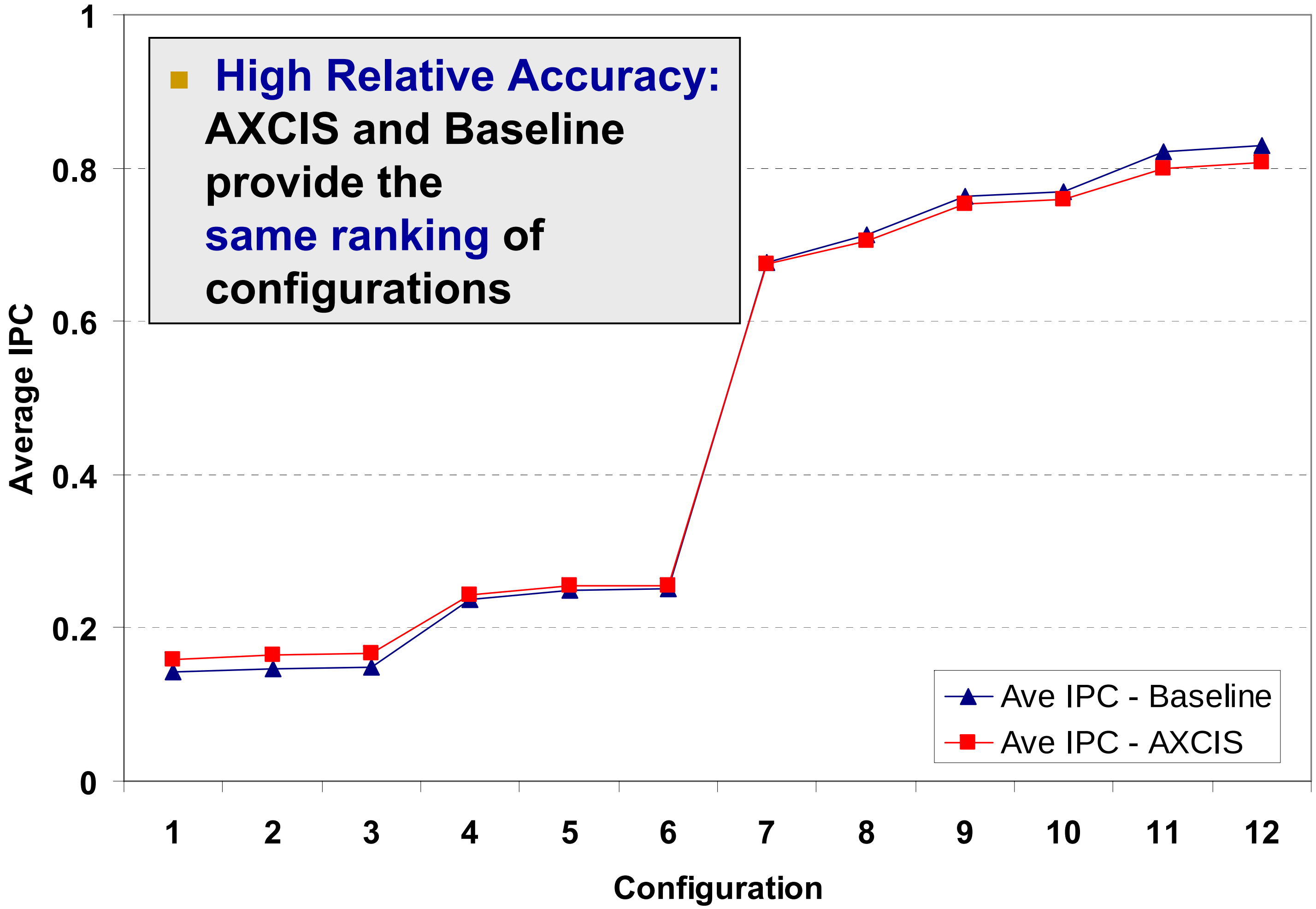
Results: Accuracy

Distribution of IPC Error in quartiles



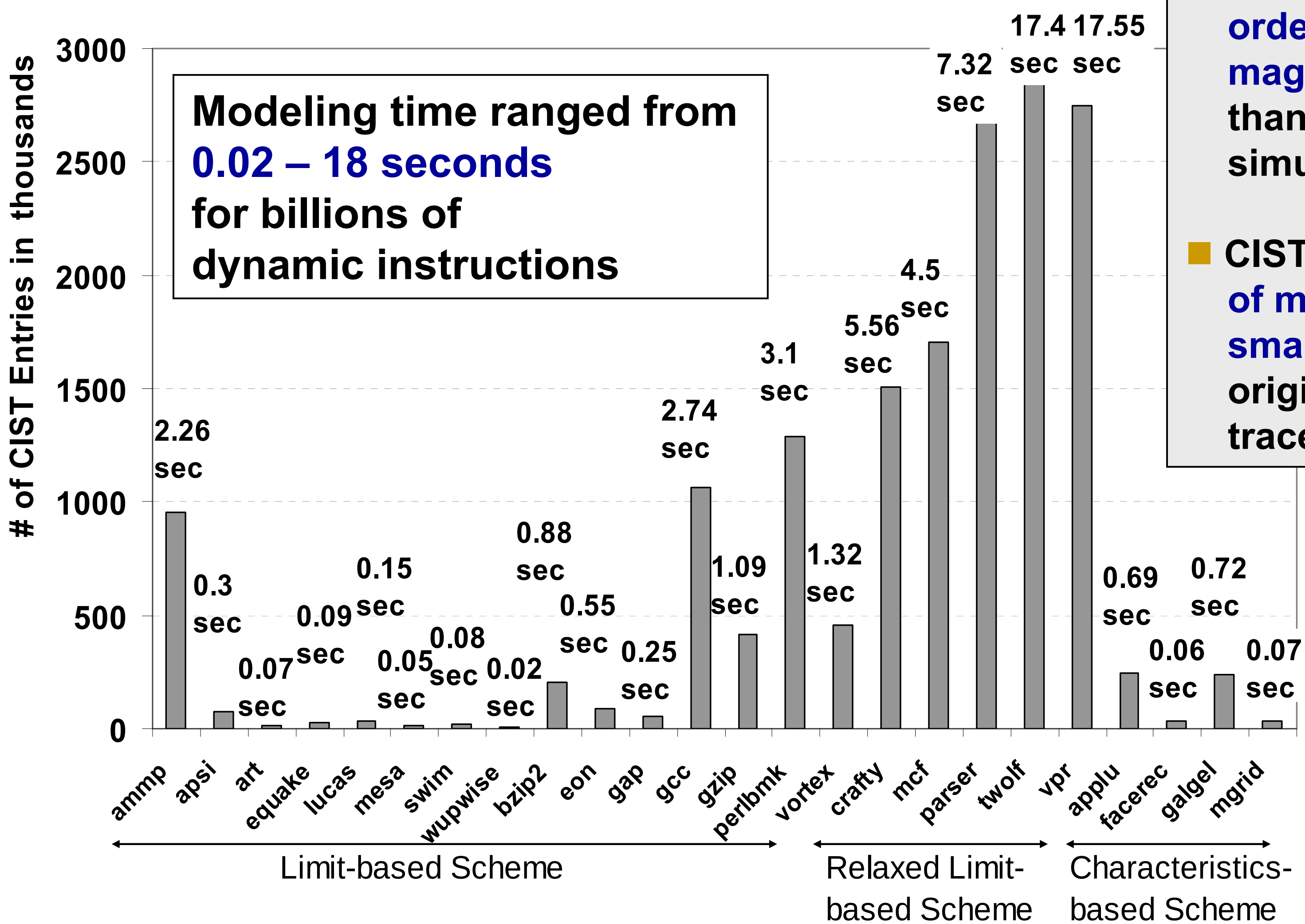
Results: Relative Accuracy

Average IPC of Baseline and AXCIS



Results: Speed

of CIST entries & modeling time



- AXCIS is over **4 orders of magnitude faster** than detailed simulation
- CISTs are **5 orders of magnitude smaller** than the original dynamic trace, on average

Discussion

- **Trade the generality of CISTs for higher accuracy and/or speed**
  - E.g. fix the issue width to 4 and explore near this design point
- **Tailor the tradeoff made between speed/compression and accuracy for different workloads**
  - **Floating point benchmarks** (*repetitive & compress well)*
    - More sensitive to any error made during compression
    - Require compression schemes with a stricter segment equality definition
  - **Integer benchmarks: (***less repetitive & harder to compress*)
    - Require compression schemes that have a more relaxed equality definition

Future Work

- **Compression Schemes:**
  - How to quickly identify the best compression scheme for a benchmark?
  - Is there a general compression scheme that works well for all benchmarks?
- **Extensions to support Out-of-Order Machines:**
  - Main ideas still apply (instruction segments, CIST, compression schemes)
  - Modify performance model to represent dispatch, issue, and commit stages within the microarchitectural state so that given some initial state & an instruction, it can calculate the next state

Conclusion

- **AXCIS is a promising technique for exploring large design spaces**
  - **High absolute and relative accuracy** across a broad range of designs
  - **Fast:**
    - 4 orders of magnitude faster than detailed simulation
    - Simulates billions of dynamic instructions within seconds
  - **Flexible:**
    - Performance modeling is independent of the compression scheme used for CIST generation
    - Vary the compression scheme to select a different tradeoff between speed/compression and accuracy
    - Trade the generality of the CIST for increased speed and/or accuracy

Acknowledgements

- This work was partly funded by the DARPA HPCS/IBM PERCS project, an NSF Graduate Research Fellowship, and NSF CAREER Award CCR-0093354.